Jakarta: A Language for Software Generators[1]

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

Jakarta is a superset of the Java language that is designed to support the GenVoca model of software generation. Among its distinguishing features are code generation, language extensibility, multi-class encapsulations, and subjectivity.

1 Introduction

Generators hold great promise in alleviating the onerous costs of producing and maintaining families of customized, high-performance applications. Generators convert declarative specifications of target applications into optimized source code. An approach that is gaining popularity in software generation is to use architectural specifications of target systems, where a specification is a composition of primitive building blocks called *components*. The evolution of application software — i.e., the ability to add new features and replace old ones — is accomplished by revising the composition of components that defines the target application and regenerating.

There are two rather different kinds of generator technologies: compositional and transformational. Both compose components in similar ways, but the nature of their components are quite different. *Compositional* components encapsulate code that applications execute at run-time. *Transformational* components encapsulate algorithms that *generate* the code that applications execute at run-time. The advantage of transformational technologies is that domain-specific optimizations can be an integral part of software generation; such optimizations are performed at application generation-time, yielding efficient source code. Compositional technologies generally don't perform domain-specific optimizations (or if they do, the optimizations are performed at application run-time, with a concomitant and sizable run-time overhead). Compositional technologies may be preferred over transformational technologies in domains where domain-specific optimizations play a minimal role in application performance or in domains where components must be composable at application run-time and cannot be limited to static compile-time compositions.

Rather different design and modeling methodologies exist for transformational generators (e.g., Draco, KIDS) and compositional generators (e.g., OO Frameworks). However, a design paradigm has emerged, called *GenVoca*, that unifies important aspects of both compositional and transformational approaches by treating them as alternative ways of implementing component models of software domains. This is possible because GenVoca components define stereotypical refinements that occur in a domain. Such refinements can be implemented as "stupid" compositional components (i.e., those that do not encapsulate domain-specific optimizations) or as "intelligent" transformational components (that do encapsulate such optimizations). Having a single modeling methodology that decomposes domains into reusable components without a priori commitments to a compositional or transformational implementation, makes GenVoca a very powerful methodology for conceptualizing generators and families of systems.

1. This research was supported in part by DARPA contract F30602-92-2-0226 and The Applied Research Laboratories at the University of Texas at Austin.

GenVoca is a novel mixture of parameterized programming, object-oriented programming, and metaprogramming (i.e., programs that manipulate other programs). It is not surprising that the key concepts of GenVoca are not supported by conventional programming languages. Consequently, these concepts must be coded (and recoded) each time a generator is built. Coding generators from scratch is not scalable or cost effective. Thus, creating a programming language that supports GenVoca (and thus simplifies the construction of GenVoca generators) would be a major advance in software generation technology. Creating such a language is the goal of our research.

Programming support for both compositional and transformational generators requires a very powerful language. The language must be extensible, i.e., it should be possible to easily add new (primitive) domain-specific data types and programming constructs. It must also support the creation and manipulation of program fragments. (Such a capability would be used to transform program fragments that reference domain-specific concepts into program fragments that directly implement these concepts). Most conventional programming languages (e.g., C, C++) provide none of these capabilities. Lisp and CLOS, in contrast, do provide powerful features for language extensibility (e.g., Lisp macros) and program fragment creation (e.g., quote and unquote). Unfortunately, Lisp and CLOS are not the programming languages that generator technologies will have their greatest impact. Extending a more conventional (i.e., imperative) object-oriented languages will have a far greater impact and influence. For similar reasons, extending a more conventional language will make the advances needed in programming languages more understandable and accessible to the communities that would most benefit from generators. So in a real sense, bringing some of the extensibility and program fragment capabilities of Lisp and CLOS to other programming languages is a necessary but not sufficient condition for popularizing generators and simplifying generator construction. Because the parsing technologies for (mainly) functional languages like Lisp and CLOS are rather different than those for imperative languages (e.g., C, C++, Java), there are major technical challenges in realizing such a transition.

Originally, we envisioned C++ as the base language for our work. However, the stunning complexity of C++ precludes extensions in any reasonable fashion. Java, on the other hand, is a streamlined object-oriented programming language that is growing in popularity and is devoid of the complexities of C++. Moreover, it supports key features (e.g., class interfaces) that is a central requirement of GenVoca. This paper outlines an extension to Java, called *Jakarta*[2], that is designed to support GenVoca. Jakarta is presently envisioned as a preprocessor to Java, but there is no a priori reason that it must remain so.

The design of Jakarta has been strongly influenced by our work on the Genesis and P2 generators, Microsoft's IP (Intentional Programming) project, and the Lisp, CLOS, RScheme and Beta programming languages.

1.1 An Overview of Jakarta Preprocessing

Jakarta parses programs in multiple phases. A Jakarta program is first parsed into an *abstract syntax tree (AST)*. Each AST node represents a primitive language construct, where child subtrees denote the parameters of that construct. (For example, an instance of an **`if-then-else`** construct would be represented by a single node with subtrees for the boolean condition, the **`then`**-action, and the **`else`**-action). During the first phase, the symbol table is populated with class definitions (e.g., exported methods and variables) and each AST node is annotated with type information. Type checking is performed (bottom-up) during this phase of parsing.

2. Jakarta is the capital of Java, Indonesia. Neat place, good people, great food.

There are three types of AST nodes: pure-Java, pure-Jakarta, and intentions. *Pure-Java* nodes are instances of constructs in the Java language. *Pure-Jakarta* nodes are instances of constructs that Jakarta has added to Java. The remaining nodes are called *intentions*. Intentions are instances of domain-specific constructs that extend the Jakarta language; they correspond roughly to instances of Lisp macros.

Once a Jakarta program is parsed into an AST, two additional phases of parsing are performed — removal of compiler directives and intension reductions — followed by code generation.

Removal of Compiler Directives. Compiler directives can be imbedded into Jakarta programs. These directives instruct Jakarta to dynamically load libraries or to call functions in these libraries to generate code (or more accurately, to generate ASTs that are substituted in place of compiler directives). An example of code generation is the instantiation of templates. Templates are compiler functions; parameters to templates are parameters to these functions. When a template function is invoked, it returns the AST of a code segment where template parameters have been inserted at designated points.

Other possible compiler directives perform global reductions or global analyses. Examples of global reductions are the application of object-oriented design patterns. A primitive design pattern, for example, can promote methods and variables of specific subclasses to their superclasses. Applying such patterns may have a global impact on a program, as many widely distributed statements in the program may be updated. Thus complex, architectural-level modifications of programs, which are highly error-prone if performed manually, can be performed correctly and automatically within Jakarta. A simple example of a global analysis would be to output a globally revised program, and to terminate Jakarta. Thus, Jakarta can be used as a general tool for metaprogramming (i.e., programs that manipulate other programs).

Upon completion of this parsing phase, the AST of the program contains no compiler directives and represents the complete source code to be processed.

Intension Reduction. The next phase deals with the reductions of intentions, which roughly corresponds to the expansion of Lisp macros. In a top-down walk of the AST, each intention is replaced by an AST of pure-Java and pure-Jakarta nodes; the AST subtree arguments of intentions may be modified in the process. Such rewrites are performed by (Jakarta) functions that implement AST “macros”; AST macros can be simple patterns in which macro arguments are inserted at designated points, or they are complex patterns that are synthesized from arbitrarily-complex analyses. (Examples of the latter are domain-specific code generators).[3]

Code Generation. A final top-down walk of the AST generates code. A function is invoked on the root of the AST to produce the text of its Java equivalent; this recursively calls print functions of the roots of each AST subtree.[4,5]

---

3. Note that compiler directives may emulate Jakarta intentions. That is, using (ugly) compiler directives to make external calls to generate code is equivalent to using syntactically-sugared extensions that trigger the same external calls. Thus, it is possible that a template capability could be achieved both by compiler directives and through intentions.

4. If Jakarta were to be implemented as a true compiler, the code generation phase would replaced by a reduction of intentions, where pure-Jakarta nodes would be intentions and their reductions would replace them with their pure-Java AST representations.

5. It may be possible to merge some of these parsing phases together (e.g., intension reduction and code generation), which may (substantially) increase the speed at which Jakarta programs are compiled.

1.2 Jakarta Extensions to Java

There are three basic extensions that Jakarta makes to Java: syntax tree constructors, compiler directives to invoke externally-defined functions/reductions, and language extensibility.

- **Surface Syntax Trees**. Jakarta provides language support for generating, parameterizing, and composing code fragments in the form of *surface syntax trees (SSTs)*. SSTs are identical to the ASTs that Jakarta uses internally during program compilation, except that type checking and symbol table information is not maintained. Constructors for SSTs are similar to the quote/unquote constructs in Lisp.
- **Compiler Directives**. Jakarta provides compiler directives that can be inserted into a program to invoke externally-defined functions during that program's compilation. As mentioned in the previous section, these functions may dynamically load libraries, perform program transformations and analyses, and generate code (e.g., instantiating templates).
- **Encapsulated Language Extensions**. Jakarta allows extensions to its grammar to support domain-specific programming constructs. Parsing these constructs will introduce intention nodes into Jarkarta ASTs. Jakarta provides capabilities for encapsulating domain-specific extensions (i.e., new grammar productions and the Jakarta libraries that reduce intensions to pure-Java code). Prime examples of extensions will be domain-specific generators. *Thus, generators for different domains can be packaged, purchased, and linked separately to Jakarta precompilers as the need arises. This will greatly encourage the proliferation of generators.*

1.3 GenVoca Extensions to Jakarta

The core features of Jakarta actually have little to do with GenVoca; GenVoca extensions will be constructed and packaged as an extension to Jakarta. There are three basic features that GenVoca makes to Jakarta:

- **Large scale components**. GenVoca components are suites of interrelated classes that cooperate as a unit (i.e., components are generalizations of OO frameworks). *The GenVoca extension provides language support for defining plug-compatible libraries of GenVoca components and their compositions.*
- **Subjectivity**. A phenomena that arises in families of related systems is that objects that are common to multiple systems often do not have the same interface. This variability of interfaces, called *subjectivity*, has a profound impact on the organization of generators and component implementations: individual components can enlarge interfaces that they share with other components by adding new methods. *The GenVoca extension provides language support for subjectivity and for viewing objects, components, and software systems through different interfaces.*
- **Version Compatibility**. Alteration of a shared interface (through the addition of new methods) normally requires recompilation of all modules that reference this interface. (A variant is known as the *fragile superclass* problem). *The GenVoca extension provides language support for altering interfaces that are shared by libraries of components without requiring their wholesale recompilation. Thus, components that implement or reference earlier versions of an interface can remain compatible with components that deal with later versions.*

1.4 Overview of Paper

This paper outlines the SST features of Jakarta and presents an overview of the GenVoca extension. The designs for language extensibility and the handling of compiler directives are not yet complete, but will be detailed in subsequent papers. A brief discussion on how extensibility and compiler directives will be achieved is included.

2 The Jakarta Language: Syntax Trees

Jakarta provides three key features for code generators: (1) creation, parameterization, and instantiation of surface syntax trees; (2) automatic name mangling of identifiers; and (3) searching syntax trees. Each of these features will be discussed in detail in the following sections.

2.1 Surface Syntax Trees

A central feature of Jakarta is support for code generation: i.e., linguistic features that enable Jakarta programs to synthesize other Jakarta programs. Let's look at a simple problem of code generation to see where the difficulties lie. Suppose we want to create a surface syntax tree for the expression **`7+x*8`**. The corresponding code to do so is:

```
AST_exp y;                        // variable of type AST expression

y = new AST_plus( new AST_const("7"), AST_times( new AST_id("x"),
    new AST_const("8") ) );
```

where **`AST_plus`** is the AST node type for the addition operation, **`AST_times`** is the AST node type for multiplication, etc. Obviously **`7+x*8`** is a trivial expression, yet constructing its SST manually by creating and linking the appropriate AST nodes is a *very* tedious and error-prone task. Moreover, the resulting code is virtually impossible to read and maintain. *Manual creation of SSTs is impractical for all but the most trivial of code fragments.*

What is needed is a generator that produces SST code directly from easily readable and maintainable specifications. Jakarta has such generator:

```
y = exp{ 7 + x * 8 }exp;
```

`exp{ … }exp` are parentheses that enclose a syntactically correct Jakarta expression whose SST is to be generated. Jakarta replaces the **`exp{ … }exp`** expression with the ugly one that nests calls to AST node constructors. The above statement assigns the root of the SST for the expression (**`7+x*8`**) to variable **`y`**.[6]

Every non-terminal production of the Jakarta grammar could serve as starting point for parsing code fragments. It turns out that only a handful of productions (such as expressions) really need to have generators. Jarkarta specifically recognizes a set of productions/AST node types. To generate ASTs that are "values" for these types, Jakarta has different kinds of "parentheses" — called *SST constructors* — that return syntactically (i.e., grammatically) correct Jakarta SSTs. The following SST constructors have been identified, although additional constructors may be added:

AST Type	Semantics	Parentheses
`AST_id`	method, class, and variable identifiers	**`id{ … }id`**
`AST_typ`	type names	**`typ{ … }typ`**
`AST_exp`	expressions	**`exp{ … }exp`**
`AST_xlst`	comma-separated list of expressions	**`xlst{ … }xlst`**
`AST_plst`	list of formal parameters	**`plst{ … }plst`**

6. Remember that SSTs are ASTs for which type checking has not been performed and linkages to the symbol table have not been made. Such checking and linkages can be performed by special methods after SST creation.

AST Type	Semantics	Parentheses
AST_stm	statements	**stm{ … }stm**
AST_mth	methods	**mth{ … }mth**
AST_cls	interface and class declarations	**cls{ … }cls**
AST_prg	program	**prg{ … }prg**
AST_tlst	list of type names	**tlst{ … }tlst**
AST_imp	list of import statements	**imp{ … }imp**
AST_mod	list of modifiers	**mod{ … }mod**
AST_vlst	list of variable declarators	**vlst{ … }vlst**
AST_vi	variable initializer	**vi{ … }vi**
AST_ai	array initializer	**ai{ … }ai**
AST_estm	expression statements separated by commas	**estm{ … }estm**
AST_cat	catch phrases	**cat { … }cat**

Here are some other examples of their use:

```
AST_id  n = id{ biff }id;     // n is AST of identifier “biff”
AST_typ t = typ{ int }typ;    // t is AST for the type “int”

                              // s is AST for statements “int i;
                              // if (foo.baz()) return 1;”
AST_stm  s = stm{ int i; if (foo.baz()) return 1; }stm;

                              // m is AST for method “foo”
                              // c is AST for class “biff”
                              // p is AST for “package don; import x;”
AST_mth  m = mth{ int foo() { return 3; }; }mth;
AST_cls  c = cls{ class biff { biff() { } }; }cls;
AST_prg  p = prg{ package don; import x; }prg;

                              // parameter and expression lists
AST_xlst xlst = xlst{ 4*7, 9*x,y }xlst;
AST_plst plst = plst{ int i, boo j, cls }plst;

                              // type, modifier and import lists
AST_tlst  t = tlst{ foo, bar, myinterface }tlst;
AST_mod   m = mod{ abstract final private }mod;
AST_imp   i = imp{ import z; import fluff; }imp;

                              // list of variables, catch list
AST_vlst  v = vlst{ a, b, c, d }vlst;
AST_estm  e = estm{ a, 34*4, i = 45, j = 56*5 }estm;
AST_cat   k = cat{ catch ( int i ) {} catch ( int j ) {} }cat;

                              // array and variable initializers
AST_ai   ai = vi{ { 4*5, $exp(foo) } }vi;
AST_vi   vi = vi{ 45*5 }vi;
```

SST constructors substantially simplify the creation of code fragments. In the following section, we show how SSTs can be parameterized and composed.

> **Implementation Notes**. The Java grammar is fairly easy to augment with SST constructors. I have the augmented grammar and have test cases to validate the design.

2.1.1 Parameterization and Composition of Abstract Syntax Trees

Code fragments can be parameterized so that other code fragments can be substituted as parameter values. Parameterizing SSTs can be done both implicitly and explicitly in Jakarta. The following sections explain both approaches.

2.1.1.1 Implicit Parameterization and Composition

Jakarta AST variables are treated specially within SST constructors: they define implicit parameters. Consider the AST variables **y** and **s**:

```
AST_exp y = exp{ foo.bar(8) }exp;

AST_stm s = stm{ if (y>4) return r; }stm;
```

When the second assignment is executed, variable **s** is assigned the SST for the statement (**if (foo.bar(8)>4) return r;**). That is, the SST "value" of variable **y** is substituted directly into the code fragment for **s**.

> **Implementation Notes**. When Jakarta parses the code fragment of an SST constructor (at code generation time), it examines each identifier. If the identifier is an AST variable, the value of that variable is substituted into the AST; if the identifier isn't an AST variable, code to generate a reference to that identifier is produced. For example, the code that Jakarta generates for statement **s** above is shown below; the value of **y** is substituted directly (because it is an AST variable) whereas a reference to identifier **r** (which is not an AST variable) is manufactured:
>
> ```
> AST_stm s = new AST_if( new AST_gtr( y, new AST_const("4") ),
>             new AST_return( new AST_id("r") ) );
> ```

As another example, the body of a loop can be constructed before the loop itself. Through implicit parameterization, both the loop and body can be composed:

```
AST_stm Body = stm{ … }stm;

AST_stm Loop = stm{ for(int i = 1; i< 30; i++) { Body; } }stm;
```

The utility of this feature becomes evident when SST constructors are used within methods that instantiate common code fragments:

```
AST_stm common( AST_exp y ) { return stm{ if (y>4) return r; }stm; };

AST_exp x1 = exp{ 5 }exp;
AST_exp x2 = exp{ 7*g }exp;

common(x1);                    // returns AST of "if(5>4) return r;"
common(x2);                    // returns AST of "if(7*g>4) return r;"
```

There are two points worth noting. First, Jakarta ensures that the SSTs it creates and composes are syntactically correct (i.e., grammatically correct). However, there is no guarantee that the resulting SSTs are semantically correct (i.e., correspond to correct ASTs). For example, suppose variable **`g`** above is a boolean. Thus, the SST produced by **`common(x2)`** — (**`if(7*g>4) return r`**) — will be syntactically correct, but semantically incorrect. The reason is that the **`common`** method only requires ASTs to be of type **`AST_exp`**, not specifically ASTs of numerical expressions.

So why can't semantic checking of SST constructors be done by Jakarta? It can once the SST of the full program has been produced. However, semantically checking isolated fragments can be problematic. Remember that programs are assembled from fragments. This means that the scoping and definition of variables within code fragments will be difficult, if not impossible to determine: SSTs that declare variable types may be created separately from SSTs that reference these variables. Only until the latter stages of program assembly where fragments are pieced together and variable scopes can be determined, can relationships between variable declarations and references be made.[7]

Limitations. The use of AST variables to "implicitly" parameterize of SST constructors is likely to be the most common way in which code fragments are defined and composed, simply because implicit parameterizations are really easy read and maintain. However, the Java grammar does not allow identifiers to appear in arbitrary places. So there are limitations where AST variables can appear. The following rules indicate the possible placement of AST variables in code fragments:

Node Type	Placement Guidelines
`AST_id`	any place where an identifier can appear
`AST_typ`	any place where a type might appear
`AST_exp`	any place where an expression might appear
`AST_xlst`	any place where argument lists might appear
`AST_plst`	any place where formal parameter lists might appear
`AST_stm`	any place where a statement might appear
`AST_mth`	any place where a method or data member declaration might appear
`AST_cls`	any place where a class might appear
`AST_prg`	any place where programs (packages, etc.) might appear

TABLE 1. Guidelines for Implicit Parameterizations of Code Fragments

2.1.1.2 Explicit Parameterization and Composition

Whenever implicit parameters can't be used or are not appropriate, explicit parameterizations or substitutions can occur via *SST escapes* **`$x(…)`**, where … is a Jakarta expression that evaluates to an AST of type **`AST_x`**. The following is an equivalent declaration of the **`Loop`** variable defined in the previous section:

```
AST_stm Body = stm{ … }stm;

AST_stm Loop2 = stm{ for( int i= 1; i< 30; i++) { $stm(Body); } }stm;
```

7. Actually, this is not quite true. Smaragdakis has added a "grouping" feature to Microsoft's IP that dynamically maintains scoping information. However, we are exploring a static solution that is similar to P2's XP preprocessor.

There are escapes for each AST node type:

Node Type	SST Escape Function
`AST_id`	`$id(AST_id i)`
`AST_typ`	`$typ(AST_typ t)`
`AST_exp`	`$exp(AST_exp e)`
`AST_xlst`	`$xlst(AST_xlst x)`
`AST_plst`	`$plst(AST_plst p)`
`AST_stm`	`$stm(AST_stm s)`
`AST_mth`	`$mth(AST_mth m)`
`AST_cls`	`$cls(AST_cls c)`
`AST_prg`	`$prg(AST_prg p)`
`AST_tlst`	`$tlst(AST_tlist t)`
`AST_imp`	`$imp(AST_imp i)`
`AST_mod`	`$mod(AST_mod m)`
`AST_vlst`	`$vlst(AST_vlst v)`
`AST_vi`	`$vi(AST_vi v)`
`AST_ai`	`$ai(AST_ai a)`
`AST_estm`	`$estm(AST_estm e)`
`AST_cat`	`$cat(AST_cat c)`

TABLE 2. AST Escapes

SSTs that are substituted by escapes are typed-checked to ensure that syntactically correct SSTs are formed. SST escapes can only appear within SST constructors.

2.1.2 Recap

With SST constructors, Jakarta offers a very powerful and easy-to-use facility for manufacturing and composing code fragments. These capabilities are analogous to Lisp's quote and unquote. However, an important practical improvement is the implicit parameterization of code fragments by AST variables. From our experience, generators often utilize code fragments that are highly parameterized. By making all parameters explicit (via SST escapes) would render the code fragments unreadable. The real benefit of the XP preprocessor of P2 was to make the most common code fragment parameters implicit, thereby making code fragments themselves easily readable and maintainable.

The next section discusses another fundamental improvement over Lisp quote and unquote facilities.

2.2 Name Mangling

A common problem in software generation is synthesizing code for some class **`K`**. Often it is the case that the interface for **`K`** has been determined, and the problem becomes one of composing code fragments to define the "body" of **`K`**, i.e., its data members and the bodies for each of **`K`**'s methods. As an example, the interface to a container of elements may be known, but now the problem is to generate an implementation for this interface.

It is also common for the "body" of **K** to be synthesized from components that encapsulate different domain-specific features. In the P2 generator, for example, there are components that encapsulate code fragments that allow elements of a container to be stored on a binary tree, doubly-linked list, etc. Thus, composing a binary tree component (called **bintree**) with itself would allow elements of a container to be simultaneously stored on two binary trees (presuming each binary tree would have a different key).

Suppose **bintree** added a data member (**int i**) to class **K**. Composing **bintree** with itself would add the data member (**int i**) twice to **K**. Thus, **K** would have two integer variables with name **i**. Clearly, this is an error. To repair it requires significant modifications to **bintree**: the names of its generated data members would need to be altered so that naming conflicts are avoided when **bintree** is composed with other components. (This naming problem is endemic to compositions of arbitrary components, and is definitely not limited to this simple example).

A common solution to this problem is to mangle identifiers by providing unique extensions to names of declarations and their references. Name mangling can be done manually, but at a significant cost. Programmers would have to create **AST_id** variables to contain mangled names; this is tedious, inefficient, and error-prone. Jakarta provides language support to solve the name mangling problem using two ideas. First, given an identifier **n**, Jakarta mangles it to **n__#**, where **__#** is called the *mangle suffix* and **#** is a unique integer (called a *mangle number*) assigned by Jakarta. Second, there can be a set of distinct code fragments that reference the same (class) variables and methods. In such cases, variable names and methods must be mangled consistently — i.e., they must be given the same mangle suffix. This is accomplished by assigning mangle suffixes to objects; code fragments that this object generates will have its method and variable names mangled with the same mangle suffix.

The **mangle** directive declares specific identifiers to be mangled within SST constructors. Below is an example of its use:

```
class X {
   mangle i, j;             // method or variable names to be mangled

   AST_exp foo() { return exp{ i + j }exp; };
   ...
}
```

The **mangle** directive tells Jakarta that all references to identifiers **i** and **j** in SST constructors within class **X** are to be mangled. Moreover, different instances of class **X** will have different mangle numbers. Consider the following:

```
X a, b;               // two different instances of X
(a.foo()).print();
(b.foo()).print();
(a.foo()).print();
```

Suppose the mangle number for **a** is **3** and for **b** is **4**. The call **a.foo()** returns the SST of the code fragment (**i__3 + j__3**), while **b.foo()** returns the SST of (**i__4 + j__4**). A repeated call to **a.foo()** reproduces the code fragment (**i__3 + j__3**).

> **Implementation Notes**. Jakarta replaces a **mangle** directive with the following **String** variable **__mangle** (which contains a unique mangle suffix), plus the list of identifiers to be mangled. (This latter list is used for mangle identifiers that are supplied only at application run-time (see Section 2.2.2)).
>
> ```
> String __mangle   = Jakarta.new_mangle_string();
> ```

```
__IdList __idlist = new __IdListNode("i", new __IdListNode("j",null));
```

When Jakarta reduces a **mangle** statement, in addition to outputting the above lines of code, it remembers the list of identifiers to be mangled. (This list is also associated with the class that is being parsed). When these identifiers are encountered within an SST constructor (at code generation time), Jakarta creates an identifier node that (when the node is printed) automatically attaches the mangle suffix to the identifier's name.

2.2.1 Indirect Mangling

It is possible that multiple Jakarta classes work cooperatively together and thus should share the same mangle information. This is accomplished by specifying the **mangle using** clause. Suppose code fragments in methods of class **Y** are to mangle the identifiers declared in class **X**. This would be declared as:

```
class Y {
   X origin;
   mangle using origin;    // specifies object from which __mangle taken

   Y(X x) { origin = x; }

   AST_exp bar() { return exp{ i * j }exp; }
   ...
}
```

In the above example, every instance of **Y** is associated with one instance of class **X**. The mangle suffix of a **Y** instance is the mangle suffix of its corresponding **X** instance. Consider:

```
Y q, r;
q = new Y(a);        // q shares mangle information of object a
(q.bar()).print();   // prints i__3 * j__3
r = new Y(b);        // r shares mangle information of object b
(r.bar()).print();   // prints i__4 * j__4
```

Implementation Notes. The **mangle using** directive is easy to implement. In AST nodes that represent identifiers to be mangled, there must be a reference to the mangle suffix. Usually, this is a reference to the **__mangle** variable. However, in the above case of **mangle using**, the mangle suffix is **origin.__mangle**. Thus, the argument of **mangle using** is the object that has the mangle suffix.

It is possible that a class **Z** might define some identifiers to mangle, and mangle (using another suffix) other identifiers. The example below illustrates this possibility:

```
class Z {
   X origin;
   mangle k, l, m;        // mangle identifiers k, l, m
   mangle using origin;   // and identifiers associated with origin

   Z(X y) { origin = y; } // constructor

   AST_exp baz() { return exp{ int k, i; }exp;
}
```

```
Z z;
z = new Z(a);
```

Suppose the mangle number for **z** is **8**. Then **z.baz()** returns the SST for (**int k__8, i__3;**).

2.2.2 Dynamic Mangling

It is possible for the names of identifiers to be manufactured at generator run-time. The **runtime mangle** statement enables identifiers to be registered at run-time and mangled. Note that manufactured identifiers can be inserted into SSTs only through identifier escapes and identifier SST variables. There is a (minor) limitation on the use of dynamic registration: *manufactured names are assumed to be distinct from identifiers that are already present in SST constructors and those already listed in* **mangle** *declarations.* Here is an example:

```
class X {
   ...
   AST_exp bop( String newSymbol ) {
      runtime mangle newSymbol;
      return exp{ $id(newSymbol) + 5 }exp;
   }

   AST_exp biff( String Symbol ) {
      return exp{ $id(Symbol) }exp; }

   AST_exp surprise( String Symbol ) {
      return exp{ aaa + $id(Symbol) }exp; }
}
```

a.bop("aaa") returns the SST for (**aaa__3+5**), where again, **__3** is the mangle suffix for **a**. A subsequent call **a.biff("aaa")** will generate the SST for (**aaa__3**), as identifier "**aaa**" will have already been registered to be mangled. Note that **a.surprise("aaa")** generates the SST for (**aaa+aaa__3**). The reason that (**aaa__3+aaa__3**) is not generated is that manufactured identifiers are assumed to have names that are distinct from all other identifiers in SST constructors. Thus, the identifier **aaa** in **exp{aaa+$(Symbol)}exp** is assumed to be distinct from any value **$(Symbol)** may subsequently assume.

> **Implementation Notes**. Jakarta translates **runtime mangle x** statements into statements that add the identifier **x** to the **IdList** of the current object. (Jakarta might check to see if the name is already present — if so, an exception is thrown). At the time an SST is being constructed (somewhere inside the code produced by Jakarta that implements an SST constructor), whenever a type or symbol escape (or AST type or AST id variable) is seen, the identifier is compared with the identifiers on the **IdList**. If the identifier is present, then its name is mangled, otherwise it is not.

2.2.3 Recap

The ability to have names mangled automatically and consistently is a fundamental need for code generators. Most name mangling can be done statically, at Jakarta application compile-time, rather than dynamically at application run-time. Static mangling makes code generators more efficient, and is consistent with the approach taken in the XP preprocessor of P2.

2.3 Searching Syntax Trees

Searching ASTs for specific code fragments will be an important operation in Jakarta. Such traversals are expressed by qualified **`foreach_node`** statements.

2.3.1 The foreach_node Statement

AST traversals are supported by **`foreach_node`**. The following counts the number of nodes in an AST rooted at node **`t`**:

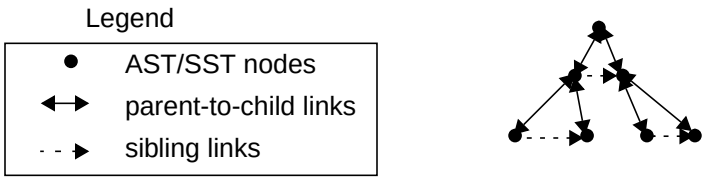
```
int count = 0;
AST_node n;

foreach_node n in t { count++; }
```

Another use for **`foreach_node`** is during the parse phase where intentions are reduced. This phase of parsing is defined by the following loop:

```
foreach_node i in Program.root where i.intention
{
  i = i.reduce();    // return root of new AST/SST
}
```

`Program.root` is a global variable that is maintained by Jakarta to contain a reference to the root of the AST of the program that is being parsed. **`i.reduce()`** unlinks the subtree rooted at **`i`** from the AST and replaces it with its reduced tree. The assignment statement is needed to set variable **`i`** to the root of this new tree, so that searching will continue with its subtrees. The **`where`** phrase imposes a qualification on **`i`** for the body of the loop to be evaluated.

Implementation Notes. To perform **`foreach_node`** traversals, AST nodes are linked directly to their parent (a null pointer designates the root of an AST), and children nodes point to their siblings. These linkages are also needed to simplify the process of replacing one AST with another. The figure below illustrates these links:



To traverse an AST, only the **`firstchild()`** and **`goright()`** operations are needed. Let **`n`** be the root of an AST that is to be traversed. The **`foreach_node`** statement:

```
foreach_node i in n where <qualification> { … /* user code */ };
```

has the reduction:

```
{ AST_node endi = n.goright();   // point to node past n (could be null)
  for (i = n; i != endi; i = i.firstchild())
  if (<qualification>)
     { … /* user code */ }
}
```

Note that the identifier **endi** in the above code fragment might need to be mangled or be renamed to an identifier that won't clash with user-defined variable names. Mangling might be needed in the case that **foreach_node** loops are nested, as there may be a confusion as to which **endi** is to be used.

The algorithm for **firstchild()** is:

- if there is a child, goto first child and return.
- if there is no child and there is a next sibling, goto next sibling (i.e., **goright()**) and return.
- otherwise, go up the AST until there is a non-null sibling pointer (or the AST root has been reached); if the root is reached, return null otherwise goto next sibling and return.

2.3.2 Qualifying Syntax Trees

Very often one is interested in examining (transforming) syntax trees that satisfy a particular qualification. Suppose one wanted to find all instances of the expression **7+8** in a program. This search could be expressed by:

```
foreach_node n in Program.root
  where (n instanceof AST_plus && n.arg[0] instanceof AST_const &&
         n.arg[0] == "7" && n.arg[1] instanceof AST_const &&
         n.arg[1] == "8")
do { … }
```

Clearly, it is hard to understand this qualification, let alone write it. In fact, this is similar to the problem we encountered earlier in writing code fragments. The solution then, as now, is to use SST generators.

Jakarta provides a powerful extension to the **foreach_node** statement for specifying qualifying patterns. The phrase **matching** allows any SST constructor to be used as a search pattern. Thus, to find all instances of the expression **7+8** would be expressed as:

```
foreach_node n in Program.root
  matching exp{ 7+8 }exp
do { … }
```

The phrase **matching exp{ … }exp** is called a *matching constructor.* There are matching constructors for all SST constructors.

A more common search pattern is knowing if a tree **x** is an instance of another tree **y**. An example would be retrieving only those nodes that correspond to **if**-statements:

```
AST_exp X;
AST_stm Y,Z;

foreach_node n in Program.root
   matching stm{ if (X) Y; else Z; }stm
do { … }
```

This example illustrates two important features. First, just as in the case of SST constructors, AST variables are treated specially within matching constructors as unbound variables. If node **n** matches the token pattern within **stm{ … }stm**, the expression is true. If true, Jakarta additionally generates code that binds the AST variables.

As an example, a possible action to take when an **if** statement is encountered is to negate the condition and swap the **Y** and **Z** actions:

```
foreach_node n in Program.root
   matching stm{ if (X) Y; else Z; }stm
do {
   n = n.replace( stm{ if (!X) Z; else Y; }stm );
}
```

The **x.replace(y)** statement replaces the subtree rooted at **x** with the subtree rooted at **y**.[8]

Limitations. At present, we will assume that all AST variables that appear in matching constructors are distinct. Thus, to search for **if** statements that have the same **then** statement as **else** statement would be expressed as:

```
foreach_node n in Program.root
   matching stm{ if (X) Y; else Z; }stm
   where Y.equals(Z)
do { … };
```

It is possible that the implementation of **foreach_node** could recognize a replication of AST variables and automatically generate the **where** phrase.

Another limitation, one that may be much more problematic is making sure Jakarta programmers understand the pattern that they are specifying. For example, to search for all interface declarations in a program may *not* be the following:

```
AST_ID X;
AST_MTH Y;

foreach_node n in Program.root
   matching cls{ interface X { Y; } }cls
do { … }
```

The problem here is that there can be optional modifiers and extends phrases. The question becomes: is this query asking for only interfaces that have *no* modifiers and extend *no* other interfaces? At present, we will assume that users will provide us with a complete match (i.e., where AST variables are present to bind to optional phrases. Thus, to search for all interface declarations would be specified by:

```
AST_MOD M; // modifier ast node
AST_TYP T; // type name AST node

foreach_node n in Program.root
   matching cls{ M interface X extends T { Y; } }cls
do { … }
```

At the time of this writing, note that the **AST_MOD** and **AST_TYP** types have note been recognized in our previous discussions. They will need to have AST constructors, escapes, too. The hope is that there won't be too many of these "additional" AST types that Jakarta will need to recognize.

As implementation and usage of **foreach_node** proceeds, we'll encounter many other "limitations". It is my hope that the basic design is sound, even though some details need to be fleshed out.

---

8. Note that when node **n** is replaced, the subtrees that will be searched next are **!X**, then **Z**, then **Y**.

Implementation Notes. The matching phrase does two things: it generates boolean conditionals for matching subtrees, and (assuming a match has been found) it generates assignment statements for each AST variable. As an example reduction, the statement:

```
foreach_node n in Jakarta.root
   matching stm{ if (X) Y; else Z; }stm
do {
   n = n.replace( stm{ if (!X) Z; else Y; }stm );
}
```

will be reduced to:

```
{ AST_node endi = Jakarta.root.goright();
  for (n = Jakarta.root; n != endi; n = n.firstchild())
  if (n instanceof AST_IF)
     { X = n.arg[0];
       Y = n.arg[1];
       Z = n.arg[2];
       { … /* user code */ }
     }
}
```

2.3.3 Matching Escapes

Whenever implicit parameters can't be used or are not appropriate, explicit variables can be declared via SST escapes. The following is an equivalent way of finding and negating if-statements:

```
AST_exp X;
AST_stm Y, Z;

foreach_node i in Program.root
   matching stm{ if ($exp(X)) $stm(Y); else $stm(Z) }stm
do {
   i = i.replace( stm{ if (!$exp(X)) $stm(Z); else $stm(Y) }stm );
}
```

Thus, SST escapes have a different interpretation within matching constructors: they define AST variables that are to be bound.

2.3.4 Limited Searches of Syntax Trees

In some cases, there may be no need to search syntax trees beyond a certain point. For example, suppose one wanted to print the names of all interfaces that were defined in program. The obvious statement to do so is:

```
AST_id X;
AST_mtd Y;

foreach_node i in Program.root
   matching cls{ class X { Y; } }cls
do { System.out.println(X); }
```

As classes are not nested, there is no need to search the subtree that encodes a class's definition. To skip the subtrees of the current node, the **`skip`** method is used:

```
foreach_node i in Program.root
   matching cls{ class X { Y; } }cls
do { System.out.println(X);
     i = skip(i);
   }
```

What the above example says is once a class node has been found (and whose name has been printed), skip its subtree(s) and goto its sibling node. (The sibling node, in this case, will be another class or interface node).

> **Implementation Notes**. **`skip()`** is a general method, like **`firstchild()`** and **`goright()`**, of all AST_node classes. There's a fairly simple trick that makes an implementation of **`n.skip()`** fast: create a dummy node when **`n.skip()`** is called; it is a terminal node whose sibling is the node identified by **`n.goright()`**. The dummy node is not really part of the tree, and can be garbage collected after it is used.

2.3.5 Recap

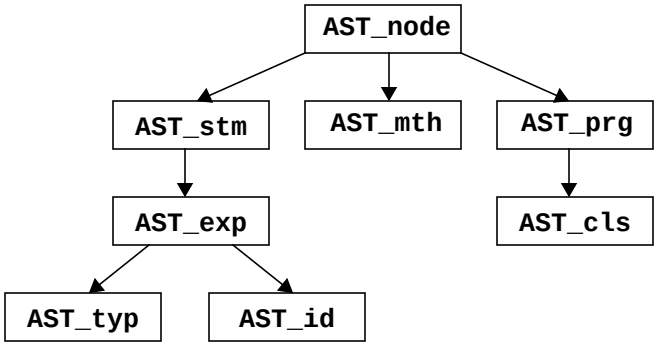
By providing language support to traverse ASTs and to find subtrees that match a given pattern, Jakarta provides basic facilities for implementing templates and other more sophisticated code generators.

2.4 Implementation Guidelines

In this section, we review a number of guidelines in the construction of Jakarta.

White Space. The Jakarta precompiler should be designed so that white space and comments in the original Jakarta program are preserved during translation. Thus, if one inputs a commented Java program as input to Jakarta, exactly the same program (by **`diff`** equivalence) will be output. This can be accomplished by pairing each token that is recognized by the Jakarta lexical analyzer with the white space, comments, (e.g., fillers) that have appeared since the last token. Thus, a data member of every **`AST_node`** is a string for white space.

Class Structure. Every AST node type will be represented as a class in an AST node hierarchy. The root of this hierarchy is **`AST_node`**. Its immediate subclasses are: **`AST_stm`**, **`AST_mth`**, and **`AST_prg`**. As the figure below shows, these classes have other subclasses. The actual number of AST node types that Jakarta will need to recognize is a open-ended now. I would expect there to be couple dozen of these types in the final version.



Class Operations. There are basic operations for every AST class: **`firstchild()`**, **`goright()`**, **`skip()`**, **`reduce()`**, **`print()`**, **`AST_qualify()`** and **`AST_print()`**. The **`reduce()`** method invokes the reduction algorithm for the corresponding AST node. Normally for all but intention nodes, **`reduce()`** does nothing but recursively call its children. **`reduce()`** is called during the intention reduction phase of parsing.

The **`print()`** method prints the Java (or Jakarta) syntax for each node. The **`AST_print()`** prints the Java code that will produce the given AST. **`AST_qualify`**() prints the code for node qualification (and is used in implementing matching constructors). Both **`print()`** and **`AST_print()`** are called during the code generation phase of parsing. Here is an example of these print methods:

```
AST_exp x,z;
z = exp{ zzz }exp;

x = exp{ 7 + z }exp;

x.print();       // outputs '7 + zzz'
x.AST_print();   // outputs 'new AST_plus(new AST_const("7"),z);'
x.AST_qualify(); // outputs 'i instanceof AST_plus &&
                 // i.arg[0] instanceof AST_const && i.arg[0].value == 7
                 // i.arg[1] instanceof AST_id && i.arg[1].value == "z"'
```

(Note: both the **`AST_print`** and **`AST_qualify`** methods really represent a transformation. **`AST_print`** could just as easily replace a "+" node with the tree represented by "**`new AST_plus(...)`**". As these transformations are so basic to Jakarta, it is not clear whether they should be given only as string generators).

3 Compiler Directives and Language Extensibility — DRAFT

At present, I'm still considering a number of options for introducing compiler directives and achieving language extensibility. Here's an outline of my current thinking.

Compiler Directives. Compiler directives will allow programs to dynamically link Jakarta with code generators, and to invoke functions of these generators. The syntax trees that are produced by these function calls will be directly inserted into the program that is being compiled. Other options would be for these functions to perform global reductions, such as implementing design patterns as discussed in Section 1.1. The problem is defining a simple metaprogramming language that can be used for such purposes.

Language Extensibility. There's a lot of work on extensible compilers. Much of this belongs to the Lisp camp, where macros are first-class language entities. Programmers can write macros (similar to Lisp macros) that can rewrite their AST/SST arguments. The core of extensible languages is defined around macros and their ability to reduce to a basic set of primitives. There are some difficulties with these approaches. Lisp, for example, has no problems because its grammar is very simple: i.e., prefix expression, that are essentially no different than ASTs (s-expressions). Adding a new node type is trivial, and its syntax is equally simple (a prefix syntax is used).

When post-fix, pre-fix, mix-fix, etc. syntaxes are combined, along with operator precedences, (as they do in imperative languages like Java), things quickly become complicated. Approaches that seem to be gaining momentum are (a) limit language extensions to a certain set of predefined "patterns" or (b) complicate the parsing by recognizing macro names and altering the parsing of the grammar to those of new extensions. So in the latter case, the base grammar is fixed, but there are "interrupts" to transfer the parsing of input tokens to other parsers.

A design approach that was followed in Genesis and P2 was that the generated code should look like code that is hand-crafted. Thus, to make compilers/languages extensible would require the resulting grammar of the extended language to look as if it were designed from scratch. This is possible if one takes the approach of "merging grammars". There is a base grammar that defines Jakarta, and there are grammar extensions, which when merged with the base grammar, yields an extended version of the Jakarta grammar.

The idea is that extensions to languages should not alter the meaning of existing constructs. (Overloading the semantics really complicates parsing, and is one of the primary reasons why C++ is so difficult to parse). So, an extension to a grammar is itself a grammar: a set of productions that either will be added to existing groups of productions or are "clean" additions. Thus, to extend Jakarta will require some grammar hacking to (a) determine which productions need to be added and (b) where these productions should be added in the existing grammar to account for precedences. By taking a "BNF-diff" of the original grammar with the extended grammar, and encapsulating the difference, the result is an "extension" to Jakarta. To install an extension, the additions are made to the grammar (and lexical analyzer), and the Jakarta precompiler is recompiled.

So language extensions won't be trivial, but doable, in Jakarta. It will lack some of the elegance that is being pursued with "macro" language approaches, but it will provide all the power needed for arbitrary language extensibility. The GenVoca extension, discussed in the next section, is being designed with this implementation approach in mind.